Journal of Physics: Conference Series **396** (2012) 052068 doi:10.1088/1742-6596/396/5/052068

Handling of network and database instabilities in CORAL

R. Trentadue, A. Valassi, A. Kalkhof

IT Department, CERN, CH-1211 Geneva 23, Switzerland

E-mail: andrea.valassi@cern.ch

Abstract. The CORAL software is widely used by the LHC experiments for storing and accessing data using relational database technologies. CORAL provides a C++ abstraction layer that supports data persistency for several back-ends and deployment models, direct client access to Oracle servers being one of the most important use cases. Since 2010, several problems have been reported by the LHC experiments in their use of Oracle through CORAL, involving application errors, hangs or crashes after the network or the database servers became temporarily unavailable. CORAL already provided some level of handling of these instabilities, which are due to external causes and cannot be avoided, but this proved to be insufficient in some cases and to be itself the cause of other problems, such as the hangs and crashes mentioned before, in other cases. As a consequence, a major redesign of the CORAL plugins has been implemented, with the aim of making the software more robust against these database and network glitches. The new implementation ensures that CORAL automatically reconnects to Oracle databases in a transparent way whenever possible and gently terminates the application when this is not possible. Internally, this is done by resetting all relevant parameters of the underlying back-end technology (OCI, the Oracle Call Interface). This presentation reports on the status of this work at the time of the CHEP2012 conference, covering the design and implementation of these new features and the outlook for future developments in this area.

1. Introduction

The Large Hadron Collider (LHC), the world's largest and highest-energy particle accelerator, started its operations in September 2008 at CERN. Huge amounts of data are generated by the four experiments installed at different collision points along the LHC ring. The largest data volumes come from the "event data" that record the signals left in the detectors by the particles generated in the LHC beam collisions and are generally stored on files. Relational database systems are commonly used instead to store several other types of data, such as the "conditions data" that record the experimental conditions at the time the event data were collected, as well as geometry data and detector configuration data. In three of the experiments, ATLAS, CMS and LHCb, conditions data and several other types of relational data are stored and retrieved from C++ or Python applications using the Common Relational Abstraction Layer (CORAL) [1, 2] software, one of the three packages jointly developed by these three experiments and the CERN IT Department within the Persistency Framework project [3, 4].

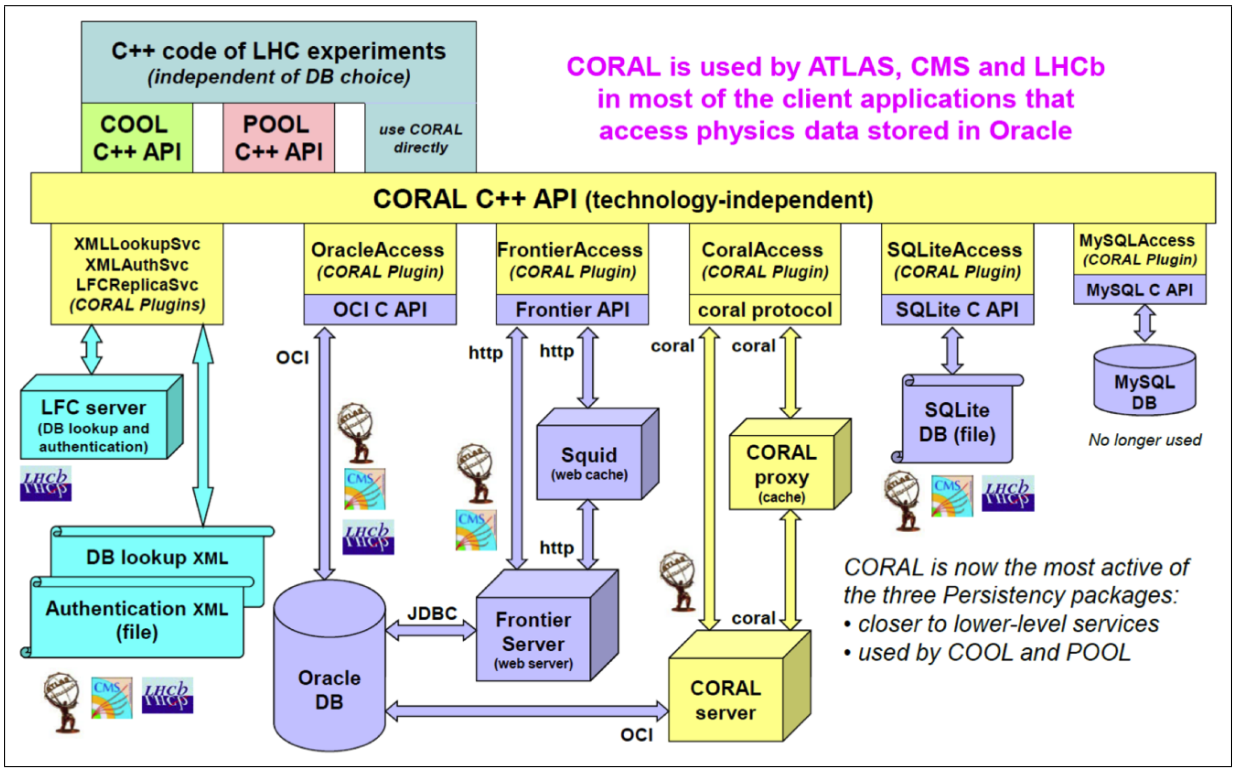

Figure 1. Back-ends supported by CORAL.

1.1. Accessing the Oracle data of the LHC experiments using CORAL

CORAL provides a C++ abstraction layer that supports data persistency for several back-ends and deployment models, as shown in figure 1: this includes local access to SQLite files, direct client access to Oracle and MySQL servers and read-only access to Oracle through the Frontier/Squid [5] and CoralServer/CoralServerProxy server/cache systems. The C++ API of CORAL consists of a set of SQL-free abstract interfaces that isolate the user code from the database implementation technology. The main advantage offered by CORAL is that users can write the same code for all back-ends, without having any detailed knowledge of the different SQL flavors, as the SQL commands specific for each back-end are executed by the relevant CORAL libraries, loaded at run-time by a dedicated plugin infrastructure.

Out of the many use cases supported in CORAL, direct access to Oracle database servers is currently the most important one, as this is the technology used by all of ATLAS [6], CMS [7] and LHCb [8] to store the master copy of their conditions databases at the CERN Tier0 site.

1.2. Handling of network and database instabilities in CORAL

As described above, the CORAL software is effectively the entry point for the LHC experiments to access from their C++ applications the data that they store using Oracle. This makes CORAL the ideal place in the software chain to implement in a common way several features optimizing data access for this relational database technology. In particular, one extremely desirable feature for CORAL would be a functionality enabling client applications to safely handle database and network instabilities, transparently recovering broken connections when possible and immediately throwing an exception in all other cases.

One implementation of this functionality was already present in CORAL since the very earliest versions of the software in 2006 [1] and had never been reported to cause any issue. As

the load on the Oracle database servers and the usage of the CORAL software increased with the gradual ramp-up of LHC operations since the end of 2009, however, several problems have been reported by the LHC experiments involving application errors, hangs or crashes after the network or the database servers became temporarily unavailable. While the exact details of these incident reports differed from case to case, it was immediately obvious that all these issues had ultimately been triggered by a glitch in the client connection to the database server, reported by the Oracle client to CORAL using a well-defined error code (most often, ORA-03113). More importantly, as the analysis of the problem progressed, it soon became apparent that in many such situations the damage was caused not only by the network glitch itself, but also by bugs and other intrinsic limitations in the handling of this external problem within the old implementation of the "reconnection" functionality in CORAL. As a consequence, a major redesign of the CORAL plugins has been implemented, with the aim of making the software more robust against these database and network glitches. The new implementation ensures that CORAL automatically reconnects to Oracle databases in a transparent way whenever possible and gently terminates the application when this is not possible.

This presentation reports on the status of this work at the time of the CHEP2012 conference. The outline of this paper is the following. An overview of connection, session and transaction management in Oracle and CORAL, and of what happens when a connection is "broken", is presented in section 2, to allow a better understanding of the problems and fixes presented in the following section. The many improvements over time since CHEP2010 in the CORAL management of lost connections are then listed and described in detail in section 3, complementing a more general paper about CORAL, COOL and POOL presented at this conference [4]. Finally, some conclusions are given in section 4.

2. Connections, sessions and transactions in Oracle and CORAL

Data access to an Oracle database from a user (client) application involves the communication of client processes on the client host with server processes on the database server host [9]. Each server process takes care of the interaction on the database host with the database "instance" and all the associated memory and storage resources on behalf of the client processes it is serving. In the Oracle language, a "connection" and a "session" indicate two different concepts which are worth explaining as they are also mapped to two different concepts (and C++ classes) in CORAL. A connection is a physical communication pathway between a client process and a database instance; this is typically established over the network as the client host and the database host are generally different. A session is a logical entity in the database instance memory that represents the state of a current user login to a database; it lasts from the time the user is authenticated (e.g. with a password) until the time the user logs out of the database or exits the application. A single connection can have zero, one or more sessions established on it. This is shown schematically in figure 2.

The client application in the client process uses the Oracle client libraries to communicate with the corresponding server process on the database host. As shown in figure 1, this is achieved in CORAL via the dedicated OracleAccess plugin, which is internally implemented using the Oracle Call Interface (OCI) [10] API and client library. Oracle connections and sessions are described in CORAL by instances of the `coral::OracleAccess:Connection` and `coral::OracleAccess:Session` classes. In the CORAL user API these classes are not directly accessible, but they are wrapped by an ad-hoc proxy class (`coral::ISessionProxy`) that takes care of establishing and releasing connections and sessions according to the user-defined configuration of the CORAL ConnectionService component. In other words, end users neither need to use OCI calls directly, nor do they need to manipulate the low-level OracleAccess CORAL classes in their code, because all the implementation details are taken care of by CORAL on their behalf.

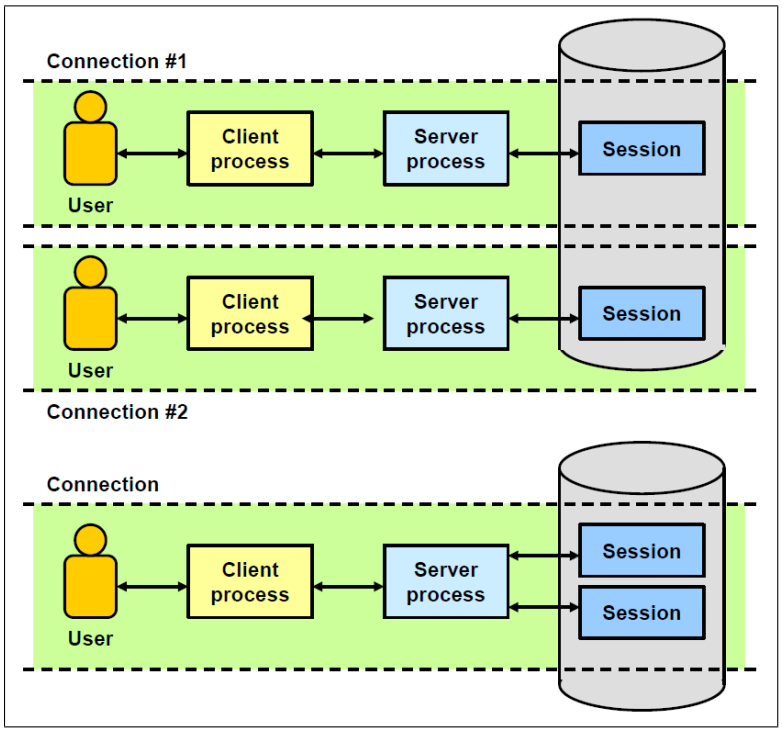

Figure 2. Client and server processes in the communication between a user application and an Oracle database [9]. Top: one session for each of two connections. Bottom: two sessions in one connection.

OCI is a C language interface and provides function calls to explicitly allocate or release "handles" to the data structures it uses internally. In the context of session and connection management, the most relevant OCI handles are the following:

- an environment handle `OCIEnv` defines a context in which all OCI functions are invoked;
- a server handle `OCIServer` identifies a connection to a database;
- a user session handle `OCISession` defines a user's roles and privileges (also known as the user's security domain) and the operational context in which the calls execute;
- a transaction handle `OCITrans` defines the transaction in which the SQL operations are performed;
- a service context handle `OCISvcCtx` contains three handles as its attributes (representing a server connection, a user session, and a transaction), which together determine the operational context for OCI calls to a server.

Within the CORAL OracleAccess plugin, each `Connection` instance owns an environment handle and a server handle, while each `Session` instance owns a service context handle, a session handle and a transaction handle, as shown schematically in figure 3. This allows CORAL to support connection sharing, i.e. to create more than one `Session` per `Connection`, which ultimately corresponds to allocating more than one OCI session handle on the same, shared, OCI server handle.

2.1. Broken connections and the ORA-03113 error code

As mentioned in section 1.2, many problems in CORAL connections to Oracle reported by the LHC experiments during the last two years could easily be identified as due to network or database instabilities because they were accompanied by a well-known Oracle error code,

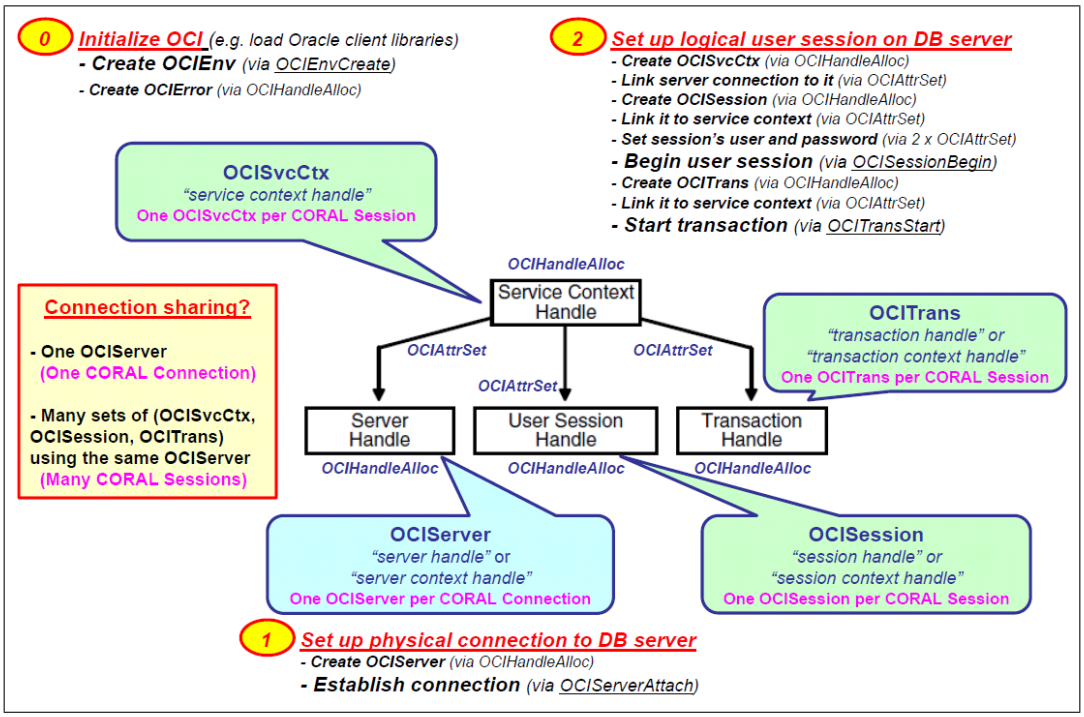

Figure 3. Implementation of CORAL connections and sessions using OCI.

ORA-03113, even if the exact details of these incident reports differ from case to case. The official description [11] of this error is:

ORA-03113: end-of-file on communication channel
Cause: `The connection between Client and Server process was broken.`
Action: `There was a communication error that requires further investigation.`

A possible cause of this issue is for instance an instability of the network, leading to a temporary glitch in the connection between the client and server. If such a network glitch occurs, our interpretation of the many tests we executed is that Oracle reacts in the following way: the Oracle server does not check whether the connection with the client is still alive, whereas it is the Oracle client that effectively probes the server as soon as the user tries to execute the first instruction after the glitch. When the Oracle client discovers that the connection to the server has been broken, any OCI function call that tries to use the broken connection (e.g. to execute an SQL statement) fails and returns the ORA-03113 error code.

The reaction of CORAL to this error is instead much simpler: both in the original version dating back from 2006 and in the new software version described in this paper, CORAL does not react to ORA-03113 errors (e.g. in the execution of an SQL statement) in any other way than by throwing an exception. This is because CORAL tries to *prevent* this error rather than *react* on it. In fact, the handling of network glitches in CORAL was (with many limitations) and continues to be (with major fixes) based on explicit probing of the client connection to the server before OCI function calls are attempted using that connection. The `OCIServerVersion` function, that is actually meant to return the version of the Oracle software version running on the server, has been used in both versions of CORAL as the probe of the client connection to the server.

We had indeed also contemplated the possibility of modifying the CORAL strategy even more to allow it to *react* on the ORA-03113 errors, rather than *prevent* them, but we estimated that this would require extensive modifications to the internal implementation code, to effectively

wrap all OCI function calls, without any significant benefit. To start with, the performance overhead of the `OCIServerVersion` function has been measured and found to be negligible, as one such function call only takes approximately 0.1 ms in our typical setup. On the other hand, this function is now used to probe the client connection to the server before essentially *every* single OCI function call using that connection is attempted, ensuring in our opinion an adequate level of protection against network glitches (ORA-03113 errors can now go unnoticed only if a network glitch takes place between the OCI probe and the "interesting" OCI call, e.g. that executing an SQL statement).

When the `OCIServerVersion` probe identifies that the connection has been lost, both the old and the new versions of CORAL react by trying to reconnect when appropriate, but with some important differences between the two, as described more in detail in section 3.5. In the old implementation, CORAL used to delete the instance of the internal C++ class representing the database connection and session (e.g. `coral::OracleAccess::Session`) and replace it by another instance of the same class, transparently for the user that only sees it wrapped by the `coral::ISessionProxy` proxy class: this was done in a generic way for all back-ends in the common ConnectionService component. The new strategy, conversely, is implemented only for the Oracle back-end within the OracleAccess plugin, and effectively moves the reconnection one level lower to the OCI layer, as it essentially consists in "refreshing" the `coral::OracleAccess::Session` by allocating new OCI handles within this instance. The new strategy is cleaner, allows a much better control over the issues that were leading to application crashes (see section 3.3) and also avoids potential issues across different back-ends by focusing only on the most relevant one (in principle, the old implementation could react to a broken Oracle connection by silently replacing it with an SQLite connection, which after many years of experience does not seem like the most appropriate course of action).

2.2. "Read-only" sessions and transactions in Oracle and CORAL

One final item of preliminary information that should be discussed before describing the recent improvements in CORAL concerns the handling of "read-only" operations. CORAL, unlike Oracle, forces users to define the "access mode" of the sessions they are about to start as either `Update` or `ReadOnly`. Within `ReadOnly` sessions, CORAL explicitly forbids any schema modification (DDL, e.g. table creation) or data modification (DML, e.g. table row insertion), throwing an exception if the user attempts to execute such an operation.

CORAL also differs from Oracle in its definition of transactions. Oracle supports three transaction types [9], or more precisely three isolation levels, as shown in table 1: in the default *Read Committed* isolation level, every query executed within a transaction sees only data committed before the query, not the transaction, began; in the *Serializable* isolation level, every query executed within a transaction sees only data committed before the transaction, not the query, began; the *Read-Only* isolation level is similar to the *Serializable* isolation level, but does not permit data to be modified in the transaction. In other words, within a *Read Committed* transaction, the same query executed at different times could in principle return different results as it represents a different database snapshot; within *Serializable* and *Read-Only* transactions, Oracle internally ensures, at a performance cost, that such a query always returns the same result as if it had been executed at the time the transaction began.

Within CORAL, as shown in table 2, the API of the `coral::ITransaction` interface only supports two transaction types, which effectively correspond to the Oracle *Read Committed* and *Read-Only* types; within the latter, CORAL does not only rely on Oracle to forbid any data modification, but also enforces this contraint internally, throwing an exception if any DML or DDL operation is attempted. Unlike Oracle, it should be noted that CORAL always expects users to explicitly start a transaction, and throws an exception if SQL statements are attempted while no transaction is active. Although this is not relevant to network glitch issues, it should

	Statement-level read consistency (read committed)	Transaction-level read consistency (serializable)
Updates allowed (RW)	*Read Committed* Isolation `OCI_TRANS_READWRITE`	*Serializable* Isolation `OCI_TRANS_SERIALIZABLE`
Updates not allowed (RO)	—	*Read-Only* Isolation `OCI_TRANS_READONLY`

Table 1. Transactions in Oracle. Three types of transactions are supported in Oracle [9, 10]. If a transaction is not started explicitly or no explicit flag is given to the `OCITransStart` call, the default *Read Committed* isolation mode is used.

	Statement-level read consistency (read committed)	Transaction-level read consistency (serializable)
Updates allowed (RW)	`ITransaction::start(false)`	—
Updates not allowed (RO)	`ITransaction::start(true)` *CORAL_ORA_SKIP_TRANS_READONLY is set*	`ITransaction::start(true)`

Table 2. Transactions in CORAL. The `coral::ITransaction` API supports two transaction types. A third "hidden" transaction type, used for instance in the CoralServer, can be enabled by setting the *CORAL_ORA_SKIP_TRANS_READONLY* environment variable.

also be noted that schema modifications via DDL statements are automatically committed in Oracle; this clearly happens also if DDL statements are issued via CORAL, which however does not handle this in any particular way, other than explicitly forbidding DDL statements in CORAL read-only sessions and transactions.

One final difference with Oracle is that CORAL foresees a third "hidden" transaction type, which is not officially supported by the CORAL C++ API but can be enabled by setting the *CORAL_ORA_SKIP_TRANS_READONLY* environment variable. If this variable is set when a session is created, calling the `coral::ITransaction::start(true)` method effectively starts an Oracle transaction with *Read Committed* isolation level, where DML and DDL updates are forbidden by CORAL. In other words, in this "read committed RO" or "non-serializable RO" transaction type, users are protected from any accidental modification of the data they are handling, but they can still see any data committed by other users in the database after their transaction began. This transaction type was added to CORAL when the CoralServer component was developed [2], because one of the use cases of the ATLAS High Level Trigger (HLT) system, where the CoralServer was first deployed, consists in the re-configuration of the HLT as new configuration parameters are inserted into the database over time, which must be performed within a single CORAL transaction due to several constraints imposed by legacy code. It should also be noted that the Frontier system effectively uses this type of transaction all the time (indepependently of any hidden environment variable setting), as it runs a *Read Committed* JDBC transaction where DML and DDL operations are forbidden by CORAL. For these and other reasons, which will also become apparent in the discussion of the network glitch in section 3.5, it is likely that this transaction type will eventually be supported also in the CORAL API.

3. Improvements in CORAL connection management since 2010

As described in section 1, the improvement and eventual reimplementation of the handling of network and database instabilities has been one of the main areas of work for CORAL in the last two years. This is a large task which has been achieved in several phases over time and is now essentially complete. In particular, identifying and fixing the CORAL bugs responsible for the observed hangs and crashes after a network glitch was the first priority for the project as soon as these issues were reported. Building on the expertise and tools developed over time, the problem was then analysed in all of its aspects, until a comprehensive strategy for a better redesign of the reconnection functionality in CORAL was identified and implemented. The various tasks achieved in this area in different phases are described in detail in the following subsections, in a roughly chronological order.

3.1. CORAL application hangs

The issue that was initially identified as the most serious one, because of both its impact (a large waste of CPU resources) and the high frequency with which it was reported by all three experiments, was an application hang accompanied by a flood of ORA-24327 ("`need explicit attach before authenticating a user`") error messages [11]. This was understood as due to a bug in the CORAL connection retrial functionality, leading to an infinite loop when retrying over and over to begin a user session over a physical connection that had been broken by a network glitch, which CORAL had failed to notice and react to. The issue was succesfully analyzed, reproduced and fixed by a patch released with CORAL 2.3.13 in December 2010.

3.2. CORAL test to reproduce database connection glitches

To reproduce the ORA-24327 problem and then all other issues, an ad-hoc test suite had to be developed: this test, written in python based on the CORAL python bindings (PyCoral) and using an ssh tunnel to simulate a network glitch, was first introduced in CORAL 2.3.13 in December 2010. It has then been significantly extended over time to cover the many other situations covered by the new CORAL reconnection mechanism and is now still routinely executed within the CORAL nightly test suite.

The generic set-up of the test is shown schematically in figure 4. On the client host, in parallel to the process running the CORAL user application, a second process is used to establish an ssh connection to a gateway host (typically, the client host itself), with the goal of creating an ssh tunnel that forwards the "listener" port of the Oracle database to a free port on the gateway host. Instead of connecting directly to the listener port of the Oracle database host, the client application establishes an OCI connection to the forwarded listener port on the gateway host. A glitch is simulated by simply killing the client ssh process, as this breaks the ssh tunnel and effectively drops the OCI connection. This method has been validated by carefully monitoring the sessions active on the database server before, during and after the glitch, using a third client

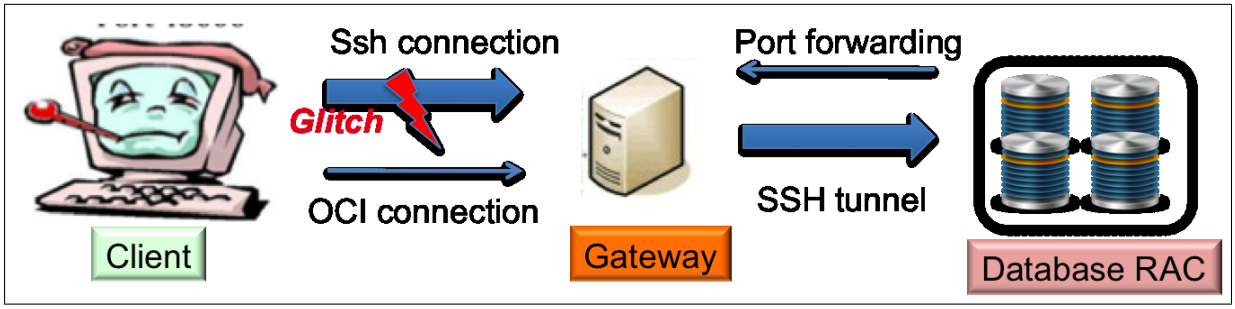

Figure 4. An ssh tunnel is used to simulate a network glitch in the connection to the database.

process directly connected to the Oracle database server via an interactive tool such as `sqlplus` or SQL Developer.

By carefully synchronising the glitch with the flow of CORAL and OCI calls in the main client process, it is then possible to test and reproduce the effect of a network glitch on the CORAL software under very many different circumstances. To reproduce the application hang described above, for instance, a glitch is triggered after establishing a connection, but before starting the user session on that connection. The same test infrastructure is now also used to validate the new CORAL implementation in many other situations, such as to study what happens if a network glitch takes place at different moments during the execution flow of an SQL query, as shown in figure 5.

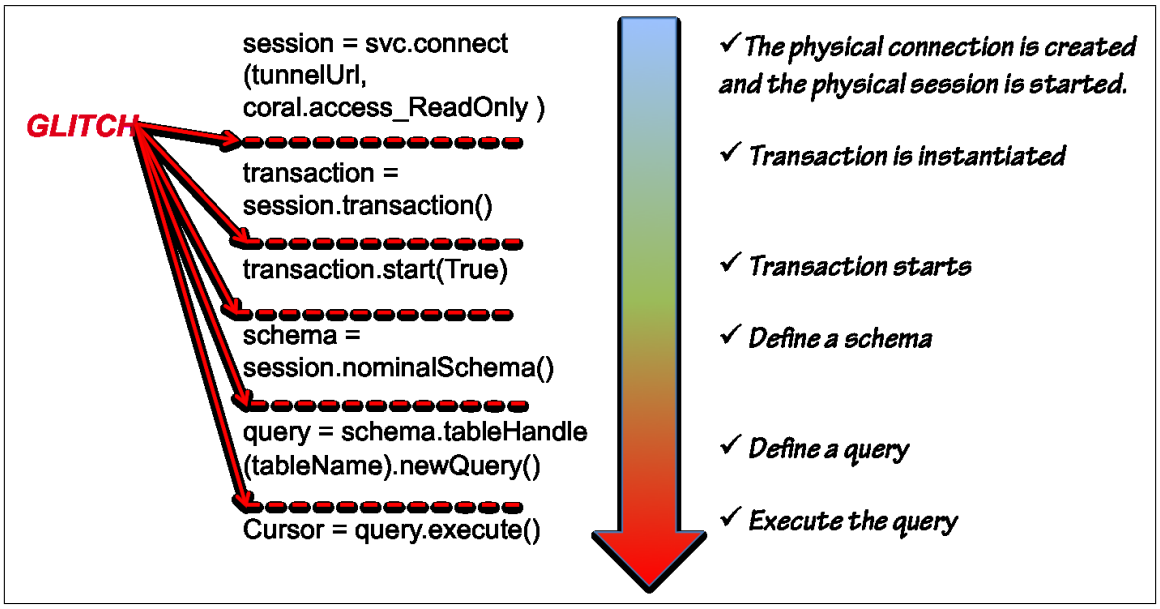

Figure 5. Workflow of the test to study the behaviour of CORAL and Oracle if a network glitch takes place at different moments during the execution flow of an SQL query.

3.3. CORAL crashes due to use of already deleted CORAL instances

The next major milestone was the analysis and fix of the CORAL crashes reported by some users in situations involving network glitches. It was soon understood that this was due to a set of bugs that affect not only the old handling of network glitches, but more generally the CORAL management of closed sessions, even when a session is closed as the result of an explicit user request to disconnect. To solve this large family of issues, it was soon understood that a major internal reimplementation of all CORAL plugins would be necessary. As mentioned before, in fact, the old CORAL implementation used to react to network glitches by deleting the `Session` instance in the relevant plugin and replacing it by another instance of the same class. However, CORAL allows users to manipulate many other classes, such as the `Query` class needed to process SQL statements and queries, which eventually need access to the properties of the `Session` instance. In the CORAL API, the lifetime of the `Query` class is unrelated to that of the associated `Session` class: in particular, the former can survive the latter if the `Session` is prematurely deleted, which could happen either in the old CORAL reconnection attempt after a network glitch, or also as a result of an explicit user disconnection. One of the problems was for instance that, within each plugin, the `Query` class was internally holding a pointer to the deleted `Session` instance, causing a segmentation fault whenever this was dereferenced.

To fix this family of issues, both the simplest ones in single threaded mode and the more complex ones that only take place in multi threaded mode, a major internal reimplementation of all CORAL plugins has been necessary. The chosen strategy essentially consists in encapsulating the status and all relevant properties of a session in an ad-hoc class (e.g. `coral::OracleAccess::SessionProperties`), which cannot go out of scope until all other instances referencing it are alive, because all such instances only possess a shared pointer to it. Out of the many patches required, those necessary to address these bugs for Oracle (in both single-threaded and multi-threaded use cases) and SQLite (in single-threaded use cases only) have been successfully completed in the CORAL 2.3.16 release in June 2011, while those for the other plugins (Frontier, MySQL and CoralServer) are being added over time, and are in some cases still pending.

3.4. Oracle client crashes due to use of already released OCI data structures

Another set of crashes reported by some users during the cleanup phase of their application (e.g. when deleting a query associated to an already deleted session), not necessarily after a network glitch, was eventually understood to be specific to the OracleAccess plugin of CORAL and to be caused by the way this uses the Oracle OCI client structures. The issue is ultimately very similar to the one described in the previous subsection: the problem is that some CORAL instances, such as again those associated to statements, queries and cursors, own OCI statement handles that internally reference other OCI connection or session handles, which may have been already released. This leads again to crashes and segmentation faults, this time deep inside the Oracle client library. One could debate whether these are CORAL bugs (i.e. Oracle undocumented "features") or Oracle client bugs (and in fact some of these issues have been reported to Oracle as such), because it is not completely clear whose responsibility it is to maintain the book-keeping of all relevant OCI data structures. In any case, major fixes have been added in the CORAL 2.3.23 release in June 2012, to make sure that no OCI session handle may be released before another OCI statement handle using it. In practice, this is simply obtained by moving any stale OCI session handles to a garbage bin within the `SessionProperties` class, delaying their release until the destruction of the latter, which is guaranteed to take place after the destruction of any CORAL `Statement` instances thanks to the shared pointer mechanism described in section 3.3.

3.5. The new implementation of the CORAL reconnection functionality

After covering all these preliminary issues, the complete reimplementation of the CORAL handling is finally described in this section. The main design ideas for this new implementation, as already described in section 2.1, are the following: within the OracleAccess plugin, CORAL issues an `OCIServerVersion` call to probe if the connection has been lost, every single time an OCI session or connection is about to be used to execute another OCI function call (in practice, this is now implemented within the relevant "getter" methods of the `coral::OracleAccess::Session` class that are responsible to return the OCI session and connection handles); if the connection has indeed been lost, whenever possible and appropriate CORAL tries to recover the broken connection, session and/or transaction, by refreshing the relevant OCI handles within the existing CORAL instances. What remains to be discussed here is how the new CORAL implementation takes the decision whether it is "possible and appropriate" to reconnect, or whether it should simply throw a `coral::ConnectionLostException` to signal that the connection has been broken by a network glitch in an unrecoverable way.

What CORAL is expected to do with each session that was interrupted by a network glitch mainly depends on the status and type of the corresponding CORAL transaction at the time the network glitch occurred. If no transaction of any type was active, CORAL tries to re-connect the broken connection and re-start the broken session, as there is no risk of data consistency

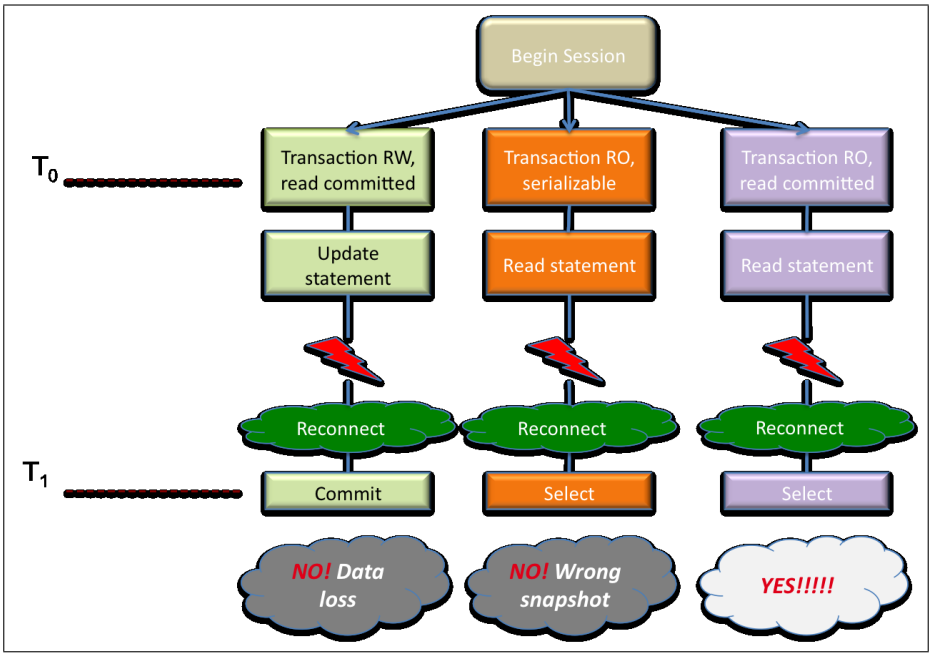

Figure 6. Expected reaction of CORAL if a network glitch breaks an active transaction.

loss. If a transaction was active at the time of the glitch, instead, CORAL reacts in different ways depending on the type of the active transaction. As shown in figure 6, the behavious of CORAL for each of the three types of transactions described in section 2.2 is the following:

- The first case is that of RW transactions (i.e. Oracle *Read Committed* transactions with updates allowed by CORAL). Suppose that the user has updated the database, for instance to add some data, but has not yet committed the transaction when the network glitch occurs. If a transparent reconnection was implemented, the user would be completely unaware of this and would then commit the transaction expecting data to be stored permanently in the database, whereas all updates have been lost. For these reasons, if a RW transaction was active when the network glitch occurred, CORAL throws an exception to avoid the risk of hidden data losses.
- The second case is that of normal RO transactions (i.e. Oracle *Read-Only* transactions with updates also forbidden by CORAL). As explained in section 2.2, this isolation level provides transaction-level read consistency, just like the *Serializable* isolation level. Suppose that the user has started the transaction at time T_0 and that the network glitch occurs at a later time T_1, but that another user in a different process has added some data in a table. Strictly speaking, the user expects that the number of rows returned by any query executed on that table will always be the number of rows that the table contained at time T_0. If a transparent reconnection was implemented, however, any query executed after time T_1 would also return the extra rows added by the second user after time T_0, breaking the expected transaction-level read consistency. For these reasons, if a (serializable) RO transaction was active when the network glitch occurred, CORAL throws an exception to avoid the risk of read inconsistency.
- The third case is that of "non-serializable" RO transactions (i.e. Oracle *Read Committed* transactions with updates forbidden only by CORAL). As explained in section 2.2, only statement-level read consistency is expected in this case. In the example above, the user always expects that a query executed after time T_1 would also return the extra rows added by

the second user after time T_0, whether or not there has been a network glitch between T_0 and T_1. For these reasons, if a "non-serializable" RO transaction was active when the network glitch occurred, and only in this case, CORAL tries to re-connect the broken connection and to re-start both the broken session and the broken transaction. It should however be noted, in this context, that CORAL does not attempt to explicitly recover a broken cursor: in other words, if a network glitch occurs in the middle of a loop over the result set of an SQL query, this eventually results in an OCI error that CORAL simply forwards to the user as a C++ exception.

The complete reimplementation of the CORAL handling of network glitches as described in this section has been completed and was finally released with CORAL 2.3.23 in June 2012. It has been succesfully tested using both the 10g and 11g versions of Oracle.

3.6. Oracle TAF (Transparent Application Failover)

While the many patches described in the above sections have successfully addressed most of the issues reported by CORAL users involving network glitches, there is still a family of issues that could not be completely reproduced, those involving for instance the ORA-25402, ORA-25405 and ORA-25408 error codes. What has been understood so far is that these errors were most likely due to broken connections that Oracle internally tried to handle using its own Transparent Application Failover (TAF) mechanism. TAF [10] is a client-side feature designed to minimize disruptions to end-user applications that occur when database connectivity fails because of instance or network failure. TAF can be implemented on a variety of system configurations including Oracle Real Application Clusters (RAC), the technology deployed on most of the CERN databases that host LHC physics data and are accessed using Oracle.

With respect to the CORAL handling of network glitches described throughout this paper, TAF could be an interesting complementary technology to address the same kind of issues, but it can also be a source of interference with the functionalities implemented in CORAL. The preliminary results of many tests we executed seem to indicate that TAF and CORAL target two different use cases: TAF can be very effective in the case of instance failures, e.g. if a full node of an Oracle cluster dies and the application is transparently redirected to another RAC node, while CORAL specifically targets network glitches that break a client-server connection without implying a full instance failure. The various ORA-2540x errors reported by our users might thus indicate issues that look somewhat similar, but are actually different from the ORA-03113 situations that are the main target of the work on CORAL described in this paper. These issues will continue to be followed up once the new CORAL release is deployed in production by the users; we also plan to continue further tests in parallel, e.g to test the behaviour of the CORAL ssh tunnel tests both in a single-node database setup and in a RAC setup using TAF.

4. Conclusion

Since 2010, several problems have been reported by the LHC experiments in their use of Oracle through CORAL, involving application errors, hangs or crashes after the network or the database servers became temporarily unavailable. A major redesign of the CORAL plugins has been implemented, with the aim of making the software more robust against these database and network glitches, and of overcoming several limitations of the previous implementation of the handling of these issues in older versions of the CORAL software. The new implementation ensures that CORAL automatically reconnects to Oracle databases in a transparent way whenever possible and gently terminates the application when this is not possible. Internally, this is done by resetting all relevant parameters of the underlying back-end technology (OCI, the Oracle Call Interface). The new implementation of this functionality is now completed and has been released with CORAL 2.3.23 in June 2012. An extensive test suite, based on ssh tunnels

to simulate network glitches, has also been developed and constitutes an integral part of the new CORAL release. Future work in this area is expected to concentrate on the Oracle TAF technology, its possible use to complement the improved CORAL functionality as well as its possible interference with it.

References

[1] I. Papadopoulos et al., *CORAL, a software system for vendor-neutral access to relational databases*, CHEP2006, Mumbai, http://indico.cern.ch/contributionDisplay.py?contribId=329&sessionId=4&confId=048
[2] A. Valassi et al., *CORAL server and CORAL server proxy: scalable access to relational databases from CORAL applications*, CHEP 2010, Taipei, http://iopscience.iop.org/1742-6596/331/4/042025
[3] A. Valassi et al., *LCG Persistency Framework (CORAL, COOL, POOL): Status and Outlook*, CHEP 2010, Taipei, http://iopscience.iop.org/1742-6596/331/4/042043
[4] R. Trentadue et al., *LCG Persistency Framework (POOL, CORAL, COOL): Status and Outlook in 2012*, CHEP2012, NY, https://indico.cern.ch/contributionDisplay.py?contribId=104&sessionId=6&confId=149557
[5] D. Dykstra, *Comparison of the Frontier Distributed Database Caching System with NoSQL Databases*, CHEP2012, NY, https://indico.cern.ch/contributionDisplay.py?contribId=218&sessionId=6&confId=149557
[6] G. Dimitrov et al., *The ATLAS database application enhancements using Oracle 11g*, CHEP2012, NY, https://indico.cern.ch/contributionDisplay.py?contribId=567&sessionId=8&confId=149557
[7] A. Pfeiffer et al., *CMS experience with online and offline Databases*, CHEP2012, NY, https://indico.cern.ch/contributionDisplay.py?contribId=163&sessionId=6&confId=149557
[8] I. Shapoval et al., *LHCb Conditions Database Operation Assistance Systems*, CHEP2012, NY, http://indico.cern.ch/contributionDisplay.py?contribId=143&sessionId=8&confId=149557
[9] Oracle Corporation, *Oracle® Database Concepts, 11g Release 2 (11.2)*, http://docs.oracle.com/cd/E14072_01/server.112/e10713.pdf
[10] Oracle Corporation, *Oracle® Call Interface Programmers' Guide, 11g Release 2 (11.2)*, http://docs.oracle.com/cd/E11882_01/appdev.112/e10646.pdf
[11] Oracle Corporation, *Oracle® Database Error Messages, 11g Release 2 (11.2)*, http://docs.oracle.com/cd/E14072_01/server.112/e10880/toc.htm

Acknowledgements

We are grateful to the users of the CORAL software in the LHC experiments for their continuous feedback and suggestions for its improvement. We warmly thank our colleagues from the Physics Database Team in CERN IT, together with the DBAs in the LHC experiments, for assisting us in understanding the subtleties of the Oracle database servers they operate.